John Hughes's "Why functional programming matters"

Marc Paterno
Neat Topics for Programmers
25 October 2011

The purpose of this "Neat Things" session

- Understand the features of lazy functional languages that Hughes argues are important.
- Discuss the utility of these features for the kinds of problems on which we work.
- See if we can obtain the same or similar advantages in C++, especially C++ 2011.

Not a negative

Features of functional languages . . .

- no assignment statements
- no side-effects in functions
- order of execution is irrelevant
- . . . yielding *referential transparency* (can replace a function call by its value)

"It is a logical impossibility to make a language more powerful by omitting features, no matter how bad they may be."

Structured programming (and more)

Structured programs are designed in a modular fashion.

- Break a large problem into smaller sub-problems
- Solve the sub-problems
- Combine the results to solve the original problem
- Modules can be written and tested independently
- The ways in which the original problem can be sub-divided depend on the ways in which one can "glue" solutions together
- Object-oriented programming introduced some new "glue" (dynamic polymorphism using classes, virtual functions)
- Generic programming introduced some new "glue" (parametric polymorphism using class and function templates)

New glue in functional languages

In lazy functional languages, Hughes identifies two important types of glue

1. Higher-order functions. Examples in the paper include:
   - *foldr*
   - *map*
   - *foldtree*
2. Lazy evaluation. Examples in the paper include:
   - *next*, in the Newton-Raphson square root implementation
   - *differentiate*, *improve* in numerical differentiation
   - *itegrate*, *super* in numerical integration

What can we learn from this for our use of C++?

- C++ 2011 provides anonymous functions (*lambda*s) and easier access to the results of type deduction (*auto*).
- We can write higher-order functions as template functions, which accept a one or more functions as function parameters.
- What is the equivalent of *lazy evaluation* for glueing functions in C++?. Can we achieve the result we want using iterators?

Hughes on lazy evaluation in imperative languages

> *"Adding lazy evaluation to an imperative notation is not actually impossible, but the combination would make the programmers life harder, rather than easier. Because lazy evaluations power depends on the programmer giving up any direct control over the order in which the parts of a program are executed, it would make programming with side effects rather difficult, because predicting in what order—or even whether—they might take place would require knowing a lot about the context in which they are embedded. Such global interdependence would defeat the very modularity that—in functional languages—lazy evaluation is designed to enhance."*

This was first written in 1984 (published in 1989). With C++ 2011 in mind as the imperative languge, is this analysis correct?

The remainder of these slides are examples from the paper, translated to Haskell, that we can use as necessary to help us understand Hughes's points.

$$
\begin{array}{lll}
\mathit{sum} & :: [\mathit{Integer}] \to \mathit{Integer} & \text{-- function type} \\
\mathit{sum}\ [\,] & = 0 & \text{-- equation (1)} \\
\mathit{sum}\ (x : \mathit{xs}) & = x + \mathit{Main.sum}\ \mathit{xs} & \text{-- equation (2)}
\end{array}
$$

We can extract the computation pattern as *comp*:

$$
\begin{array}{ll}
\mathit{comp} & :: (a \to b \to b) \to b \to [a] \to b \\
\mathit{comp}\ f\ \mathit{init}\ [\,] & = \mathit{init} \\
\mathit{comp}\ f\ \mathit{init}\ (x : \mathit{xs}) & = f\ x\ (\mathit{comp}\ f\ \mathit{init}\ \mathit{xs})
\end{array}
$$

This function appears in the Haskell Prelude, under the name *foldr*.

We could have defined *sum* using the Prelude function *foldr*:

$$\begin{aligned} &\mathit{sum2} :: [\mathit{Integer}] \to \mathit{Integer} \\ &\mathit{sum2} = \mathit{foldr}\ (+)\ 0 \end{aligned}$$

Then we get $\mathit{sum2}\ [1, 2, 3, 4] = 10$

Using *foldr* we can define other functions:

$$\begin{aligned} \mathit{myproduct} &= \mathit{foldr}\ (*)\ 1 \\ \mathit{anytrue} &= \mathit{foldr}\ (\vee)\ \mathit{False} \\ \mathit{alltrue} &= \mathit{foldr}\ (\wedge)\ \mathit{True} \end{aligned}$$

The function *myproduct* is of type
$\mathit{myproduct} :: [\mathit{Integer}] \to \mathit{Integer}$; this was automatically determined by Haskell as the most general type consistent with the definition. *myproduct* $[1, 2, 3, 4]$ evaluates to 24.

Pages 5-6

Although Haskell can almost always deduce a function type from its definition, I like to provide them explicitly.

- It is useful documentation.
- It helps make sure the function type is what I expect it to be.

$$
\begin{aligned}
&\mathit{append} \quad :: [a] \to [a] \to [a] \\
&\mathit{append}\ a\ b = \mathit{foldr}\ (:)\ b\ a
\end{aligned}
$$

(:) is the function form of the colon operator; its type is $(:) :: a \to [a] \to [a]$. The name *length* is already used in the Haskell Prelude, so we use *mylength*:

$$
\begin{aligned}
&\mathit{mylength} = \mathit{foldr}\ \mathit{count}\ 0 \\
&\mathit{count} \quad :: a \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{count}\ a\ n = n + 1
\end{aligned}
$$

Haskell allows defining "local" names with a **where** clause:

$$\begin{aligned} &\mathit{doubleall} = \mathit{foldr}\ \mathit{doubleandcons}\ [\,] \\ &\quad \textbf{where}\ \mathit{doubleandcons}\ n\ \mathit{lst} = (2 * n) : \mathit{lst} \end{aligned}$$

$$\begin{aligned} &\mathit{doubleandcons} = \mathit{fandcons}\ \mathit{double} \\ &\quad \textbf{where}\ \mathit{double}\ n \qquad\quad = 2 * n \\ &\qquad \mathit{fandcons}\ f\ \mathit{elem}\ \mathit{lst} = (f\ \mathit{elem}) : \mathit{lst} \end{aligned}$$

$$\mathit{fandcons}\ f = (:) \circ f$$

Haskell deduces the type: $\mathit{fandcons} :: (a \to b) \to a \to [b] \to [b]$.

$$\mathit{doubleall2} = \mathit{foldr}\ ((:) \circ \mathit{double})\ [\,] \quad \textbf{where}\ \mathit{double}\ x = 2 * x$$

or, using Haskell's lambda function notation,

$$\mathit{doubleall3} = \mathit{foldr}\ ((:) \circ \lambda x \to 2 * x)\ [\,]$$

Without LaTeX's fancy formatting, that looks like:

```
doubleall3  =  foldr ((:) . \x -> 2*x) []
```

The simplest definition for *doubleall* would be:

$$\mathit{doubleall4} = \mathit{map}\ (2*)$$

Page 6 (cont'd again)

map is from the Haskell Prelude, but we could have written it ourselves:

$$map2\ f = foldr\ ((:) \circ f)\ [\,]$$

Haskell tells us the type of this is: $map2 :: (a \to b) \to [a] \to [b]$.

Tree[1] is a parameterized type. We could use the Standard Library *Data.Tree*, which is slightly more general, because it can be empty, but let's define it ourselves:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Tree}\ a\ [\mathit{Tree}\ a]\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})$$

The **deriving** (*Eq*, *Show*, *Read*) makes the compiler generate some useful code for us. We can create the tree in the diagram on page 7:

$$\begin{aligned} \mathit{four} &= \mathit{Tree}\ 4\ [\,] \\ \mathit{two} &= \mathit{Tree}\ 2\ [\,] \\ \mathit{three} &= \mathit{Tree}\ 3\ [\mathit{four}] \\ z &= \mathit{Tree}\ 1\ [\mathit{two}, \mathit{three}] \end{aligned}$$

The value of *z* is *Tree* 1 [*Tree* 2 [], *Tree* 3 [*Tree* 4 []]].

[1] This type is sometimes called a *Rose tree*.

Page 7 (cont'd)

N.B.: Hughes's paper went through several revisions and changes in notation; the one we have has an error! Here's the corrected version, also translated into Haskell.

```
  -- f is applied to Nodes, g is applied to sub-Trees
foldt  f g acc (Tree lbl ts) = f lbl (foldt' f g acc ts)
foldt' f g acc (t : ts)      = g (foldt f g acc t) (foldt' f g acc ts)
foldt' f g acc []            = acc
```

This gives us types:
$foldt :: (t \to t2 \to t1) \to (t1 \to t2 \to t2) \to t2 \to Tree\ t \to t1$
$foldt' :: (t \to t2 \to t1) \to (t1 \to t2 \to t2) \to t2 \to [Tree\ t] \to t2.$
We can use this:

$$sumtree = foldt\ (+)\ (+)\ 0$$

which then gives *sumtree z* equals 10.

Page 7 (cont'd again)

We can extract all the labels

$$\mathit{labels} = \mathit{foldt}\ (:)\ \mathit{append}\ [\,]$$

Then $\mathit{labels} :: \mathit{Tree}\ t \to [t]$ and $\mathit{labels}\ z = [1, 2, 3, 4]$. And we can define the equivalent of *map* for Tree:

$$\mathit{maptree}\ f = \mathit{foldt}\ (\mathit{Tree} \circ f)\ (:)\ [\,]$$

so that $\mathit{maptree}\ (*10)\ z$ yields
$\mathit{Tree}\ 10\ [\,\mathit{Tree}\ 20\ [\,],\ \mathit{Tree}\ 30\ [\,\mathit{Tree}\ 40\ [\,]]]$.

Infinite sequences

Hughes's function *repeat* is known in the Haskell Prelude as *iterate*:

$$
\begin{aligned}
&\text{-- iterate f a} = \text{a} : \text{iterate f (f a)} \\
&\mathit{henon\_map}\ (x, y) = (y + 1 - 1.4 * x * x, 0.3 * x) \\
&\mathit{infinite\_list} \qquad = \mathit{iterate}\ \mathit{henon\_map}\ (0, 0)
\end{aligned}
$$

Note that we have *defined* an infinite list, but we have not *calculated* any elements of it. Calculation doesn't happen until we write code that actually evaluates an entry, such as *take* 3 *infinite_list* which evaluates to $[(0.0, 0.0), (1.0, 0.0), (-0.3999999999999999, 0.3)]$, or *infinite_list* !! 20 which is $(0.34951464064152427, 0.19730159602884131)$ or *infinite_list* !! 1000 which is $(-0.6277258392467069, 0.3097851907270172)$

Newton-Raphson

```
next n x                  = (x + n / x) / 2.0
within eps (a : b : rest) = if abs (a − b) ⩽ eps
                            then b
                            else within eps (b : rest)
  -- a0 is our initial guess
  -- eps is the absolute tolerance to within which
  -- we want the answer
absqrt' a0 eps n          = within eps (iterate (next n) a0)
absqrt                    = absqrt' (1.0) (1.0e − 6)
```

Then *absqrt* 4 yields 2.000000000000002, which is off by $2.220446049250313\mathrm{e}-15$ (absolute error), which is less than $1.0\mathrm{e}-6$.

NR with relative error bound

$$
\begin{aligned}
\mathit{relative}\ \mathit{eps}\ (a : b : \mathit{rest}) &= \textbf{if}\ \mathit{abs}\ (a\ /\ b - 1) \leqslant \mathit{eps} \\
&\qquad\qquad\qquad\quad \textbf{then}\ b \\
&\qquad\qquad\qquad\quad \textbf{else}\ \mathit{relative}\ \mathit{eps}\ (b : \mathit{rest}) \\
\mathit{relsqrt'}\ \mathit{a0}\ \mathit{eps}\ n &= \mathit{relative}\ \mathit{eps}\ (\mathit{iterate}\ (\mathit{next}\ n)\ \mathit{a0}) \\
\mathit{relsqrt} &= \mathit{relsqrt'}\ (1.0)\ (1.0\mathrm{e}-6)
\end{aligned}
$$

Then *relsqrt* 4 yields 2.000000000000002, which is off by $1.1102230246251565\mathrm{e}-15$ (relative error), which is less than $1.0\mathrm{e}-6$.

Note that we didn't have to rewrite the part of the code that generates the sequence of approximations.

A step that Hughes did not do

We can factor out more commonality:

$$
\begin{aligned}
&\mathit{finder}\ f\ \mathit{eps}\ (a : b : \mathit{rest}) = \textbf{if}\ (f\ a\ b) \leqslant \mathit{eps} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ b \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{finder}\ f\ \mathit{eps}\ (b : \mathit{rest}) \\
&\mathit{within}'' = \mathit{finder}\ (\lambda a\ b \to \mathit{abs}\ (a - b)) \\
&\mathit{relative}'' = \mathit{finder}\ (\lambda a\ b \to \mathit{abs}\ (a\ /\ b - 1)) \\
&\mathit{gensqrt}\ \mathit{errfunc}\ \mathit{a0}\ \mathit{eps}\ n = \mathit{errfunc}\ \mathit{eps}\ (\mathit{iterate}\ (\mathit{next}\ n)\ \mathit{a0}) \\
&\mathit{absqrt}'' = \mathit{gensqrt}\ \mathit{within}'' \\
&\mathit{relsqrt}'' = \mathit{gensqrt}\ \mathit{relative}''
\end{aligned}
$$

Even more refactoring

With *finder* defined as above, we could write

$$
\begin{aligned}
&\mathit{gensqrt'}\ f\ a0\ eps\ n = \mathit{finder}\ f\ eps\ (\mathit{iterate}\ (\mathit{next}\ n)\ a0)\\
&\mathit{absqrt'''} = \mathit{gensqrt'}\ (\lambda a\ b \to \mathit{abs}\ (a - b))\ (1.0)\ (1.0\mathrm{e} - 6)\\
&\mathit{relsqrt'''} = \mathit{gensqrt'}\ (\lambda a\ b \to \mathit{abs}\ (a\ /\ b - 1))\ (1.0)\ (1.0\mathrm{e} - 6)
\end{aligned}
$$

Then *relsqrt'''* 4 yields 2.000000000000002, which is off by $1.1102230246251565\mathrm{e} - 15$ (relative error), which is less than $1.0\mathrm{e} - 6$. Generalize one more step:

$$
\begin{aligned}
&\mathit{generate\_fcn}\ nxt\ f\ a0\ eps\ n = \mathit{finder}\ f\ eps\ (\mathit{iterate}\ (nxt\ n)\ a0)\\
&\mathit{gensqrt''} = \mathit{generate\_fcn}\ (\lambda n\ x \to (x + n\ /\ x)\ /\ 2.0)\\
&\mathit{gencbrt''} = \mathit{generate\_fcn}\ (\lambda n\ x \to (2 * x + n\ /\ (x * x))\ /\ 3.0)\\
&\mathit{relsqrt''''} = \mathit{gensqrt''}\ (\lambda a\ b \to \mathit{abs}\ (a\ /\ b - 1))\ (1.0)\ (1.0\mathrm{e} - 6)\\
&\mathit{abcbrt''''} = \mathit{gencbrt''}\ (\lambda a\ b \to \mathit{abs}\ (a - b))\ (1.0)\ (1.0\mathrm{e} - 6)
\end{aligned}
$$

And *abcbrt''''* 27.0 is 3.0000000000000977, with (absolute) error: $9.769962616701378\mathrm{e} - 14$.

What have we achieved?

- We have produced a function *generate_fcn*, that takes
  1. a function *nxt* that generates approximation $n + 1$ from approximation n,
  2. a function f that computes the measure of disagreement between its arguments (we have used absolute and relative errors)
  3. an initial estimate *a0*
  4. the maximum permissible value *eps* of the measure of disagreement
  5. the point n at which the function we're approximating is to be evaluated.
- This function can be used to create other functions, given that we have a way to (*e.g.* Newton's method) of getting from one step in the iterative approximation to the next.

Pages 12-13

$$\mathit{easydiff}\ f\ x\ h = (f\ (x + h) - f\ x)\ /\ h$$

where $\mathit{easydiff} :: \mathit{Fractional}\ a \Rightarrow (a \to a) \to a \to a \to a$

$$\begin{aligned}\mathit{differentiate}\ h0\ f\ x &= \mathit{map}\ (\mathit{easydiff}\ f\ x)\ (\mathit{iterate}\ \mathit{halve}\ h0)\\ \mathit{halve}\ x &= x\ /\ 2\end{aligned}$$

The derivative of the function f at point x can be computed by *within eps* (*differentiate h0 f x*). We can accelerate the convergence of the sequence, introducing

$$\begin{aligned}&\mathit{elimerror}\ n\ (a : b : \mathit{rest}) =\\ &\quad (b * (2 \uparrow n) - a)\ /\ (2 \uparrow n - 1) : (\mathit{elimerror}\ n\ (b : \mathit{rest}))\end{aligned}$$

where $\mathit{elimerror} :: (\mathit{Integral}\ b, \mathit{Fractional}\ a) \Rightarrow b \to [a] \to [a]$

$$
\begin{aligned}
\mathit{order}\ (a : b : c : \mathit{rest}) &= \mathit{round}\ (\mathit{log2}\ ((a - c)\ /\ (b - c) - 1)) \\
\mathit{log2}\ x &= \mathit{logBase}\ 2\ x \\
\mathit{improve}\ s &= \mathit{elimerror}\ (\mathit{order}\ s)\ s
\end{aligned}
$$

where $\mathit{improve} :: (\mathit{RealFrac}\ a, \mathit{Floating}\ a) \Rightarrow [a] \rightarrow [a]$
Note that most Haskell experts would write

$$\mathit{log2}' = \mathit{logBase}\ 2$$

rather than what we have above.

Page 13, cont'd

$$
\begin{aligned}
\mathit{super}\ s &= \mathit{map}\ \mathit{second}\ (\mathit{iterate}\ \mathit{improve}\ s) \\
\mathit{second} &= \mathit{head} \circ \mathit{tail} \\
\mathit{deriv}\ \mathit{eps}\ \mathit{h0}\ f\ x &= \mathit{within}\ \mathit{eps}\ (\mathit{super}\ (\mathit{differentiate}\ \mathit{h0}\ f\ x))
\end{aligned}
$$

where

$\mathit{deriv} :: (\mathit{RealFrac}\ a, \mathit{Floating}\ a) \Rightarrow a \to a \to (a \to a) \to a \to a$

Note that Hughes forgets to define *addpair*

$$
\begin{aligned}
&\mathit{addpair}\ (x, y) &&= x + y \\
&\mathit{easyintegrate}\ f\ a\ b &&= (f\ a + f\ b) * (b - a)\ /\ 2 \\
&\mathit{integrate}\ f\ a\ b &&= (\mathit{ei}\ f\ a\ b) : \\
&&&\quad (\mathit{map}\ \mathit{addpair}\ (\mathit{zip}\ (\mathit{integrate}\ f\ a\ \mathit{mid}) \\
&&&\qquad (\mathit{integrate}\ f\ \mathit{mid}\ b))) \\
&&&\quad \textbf{where}\ \mathit{mid} = (a + b)\ /\ 2 \\
&&&\qquad \mathit{ei} = \mathit{easyintegrate}
\end{aligned}
$$

Then *easyintegrate sin* $0\ 1$ yields 0.42073549240394825 (the correct result is 0.45969769413186028.

$$
\begin{aligned}
&\mathit{integrate2}\ f\ a\ b = \mathit{integ}\ f\ a\ b\ (f\ a)\ (f\ b)\\
&\mathit{integ}\ f\ a\ b\ fa\ fb = ((fa + fb) * (b - a)\ /\ 2) :\\
&\qquad\qquad\qquad\qquad \mathit{map}\ \mathit{addpair}\ (\mathit{zip}\ (\mathit{integ}\ f\ a\ m\ fa\ fm)\\
&\qquad\qquad\qquad\qquad\quad (\mathit{integ}\ f\ m\ b\ fm\ fb))\\
&\quad \textbf{where}\ m = (a + b)\ /\ 2\\
&\qquad fm \quad\ = f\ m
\end{aligned}
$$

where *integrate2* :: *Fractional* $a \Rightarrow (a \to a) \to a \to a \to [a]$
take 3 (*integrate2 sin* $0\ 1$) yields
$[0.42073549240394825, 0.45008051550407563, 0.4573009375715021]$
within $1.0\mathrm{e}-6$ (*integrate2 sin* $0\ 1$) yields 0.4596975479978896

Performance of super *with* integrate2

```
*Main> let ans = 1.0 - cos(1.0)
*Main> :set +s
*Main> ans - within 1.0e-6 (integrate2 sin 0 1)
1.4613397064655587e-7
it :: Double
(0.01 secs, 3697892 bytes)
*Main> ans - within 1.0e-6 (super (integrate2 sin 0 1))
-3.655964420090640 5e-13
it :: Double
(0.01 secs, 3194080 bytes)
*Main> ans - within 1.0e-12 (integrate2 sin 0 1)
1.3933298959045715e-13
it :: Double
(5.56 secs, 710636016 bytes)
*Main> ans - within 1.0e-12 (super (integrate2 sin 0 1))
-1.1102230246251565e-16
it :: Double
(0.01 secs, 3200928 bytes)
```